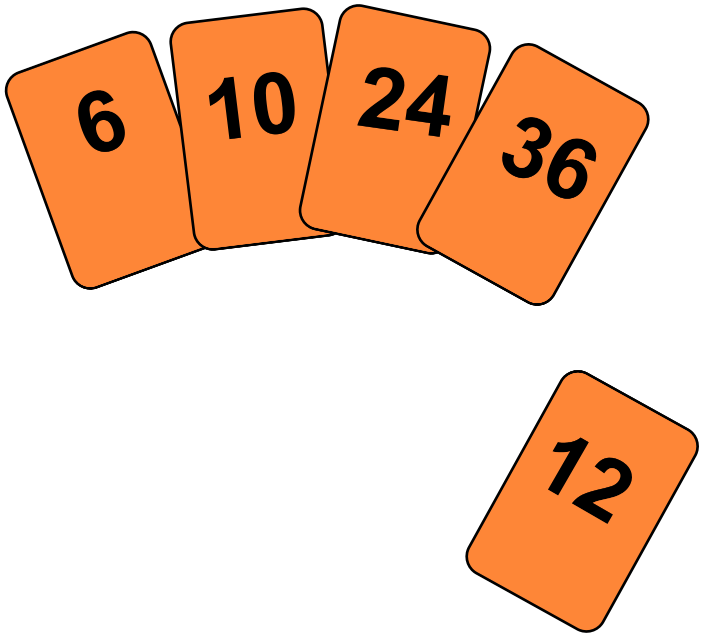
Insertion Sort

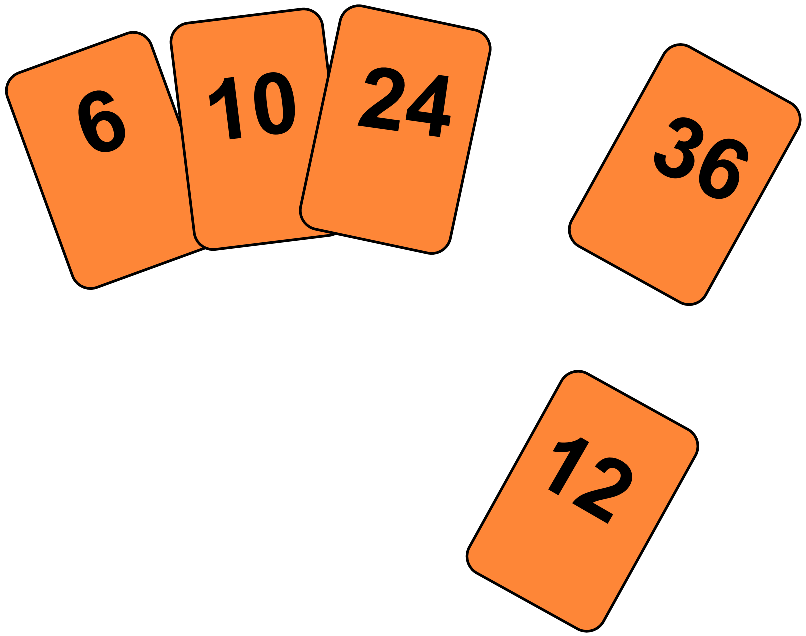
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

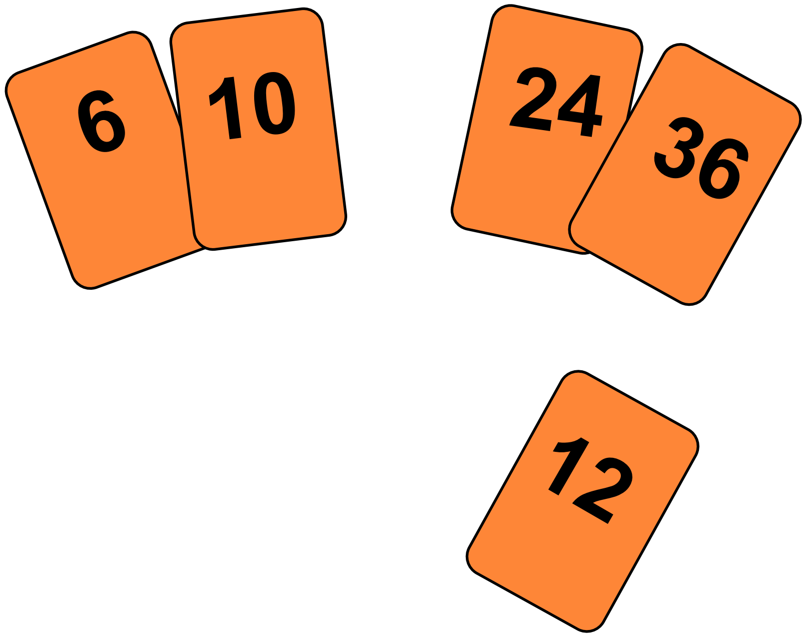


To insert 12, we need to make room for it by moving first 36 and then 24.

INSERTION SORT



INSERTION SORT



INSERTION SORT

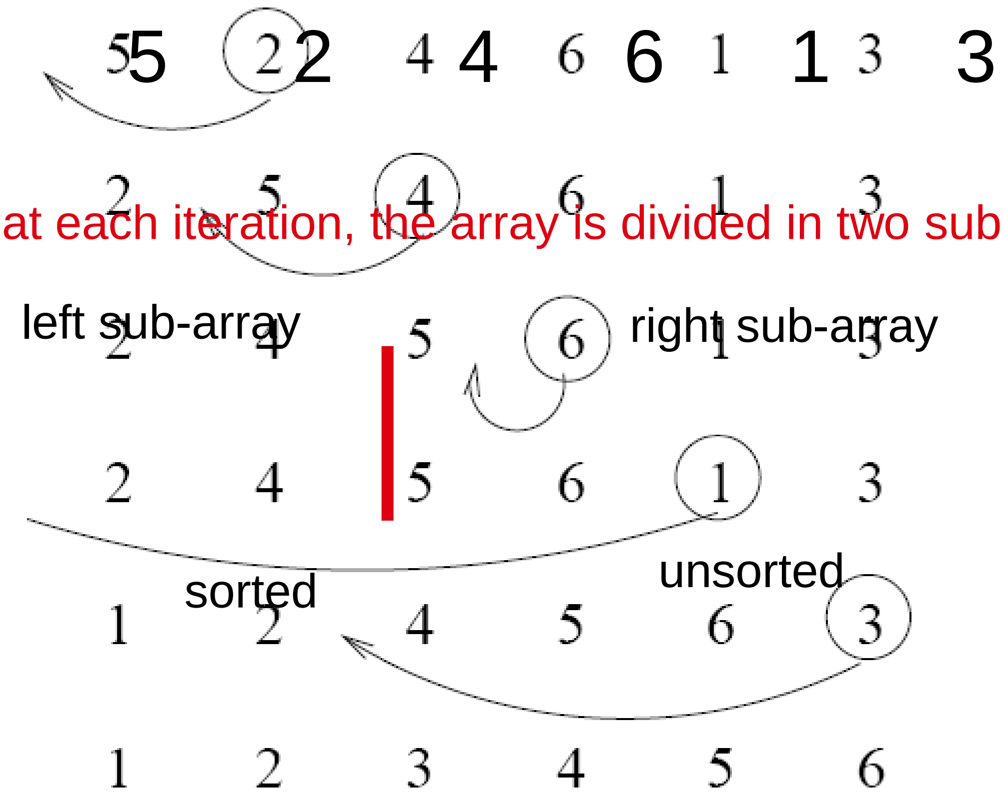
input array

5 2 4 6 1 3

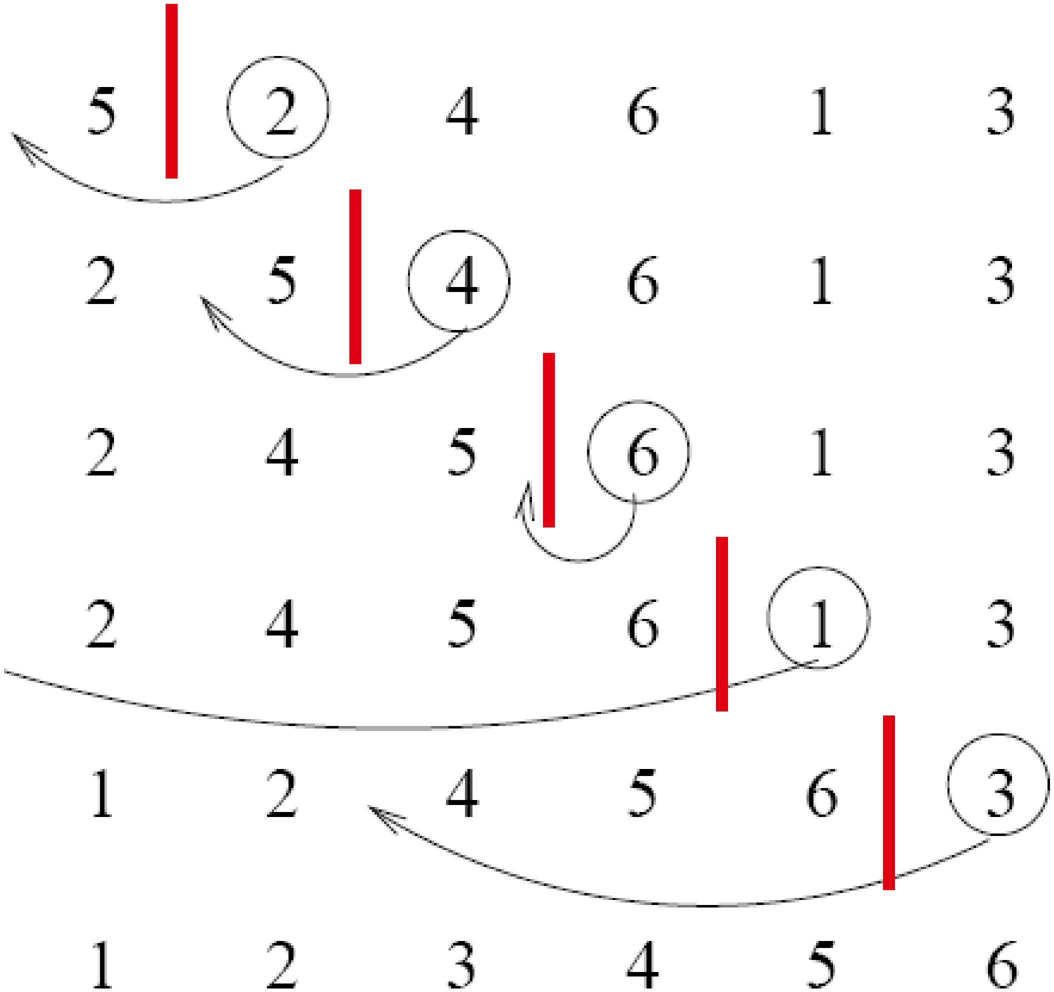
at each iteration, the array is divided in two sub-arrays:

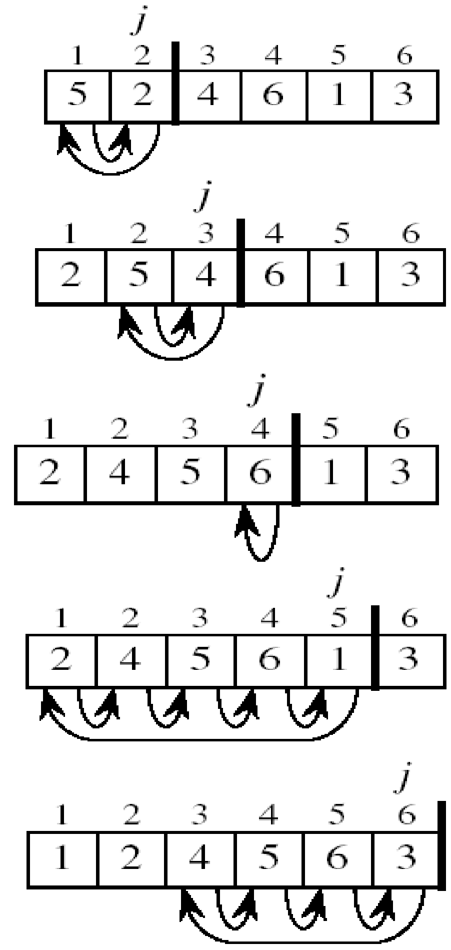
left sub-array right sub-array

sorted unsorted



Insertion Sort





INSERTION-SORT

Alg.: INSERTION-SORT*(A)*

1	2	3	4	5	6	7	8
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

key

for j ← 2 **to** n

  do key ← *A*[j]

    Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

    ▷ i ← j - 1

    while i > 0 and *A*[i] > key

      do *A*[i + 1] ← *A*[i]

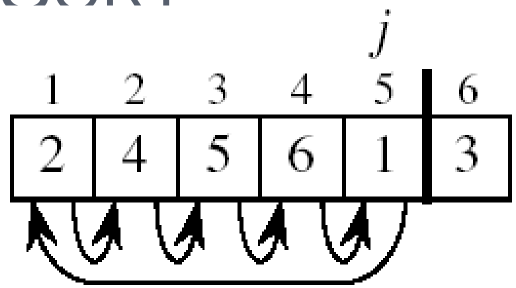
        i ← i - 1

    A[i + 1] ← key

- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT(*A*)



do key ← A[j]

Insert A[j] into the sorted sequence A[1 . . j -1]

i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i - 1

A[i + 1] ← key

Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

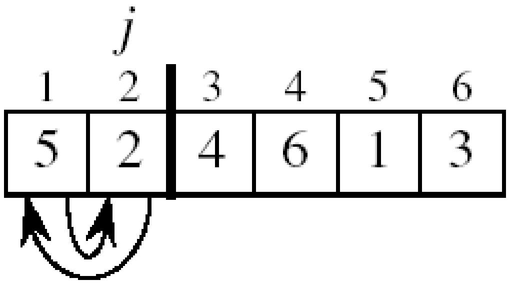
Proving Loop Invariants

- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

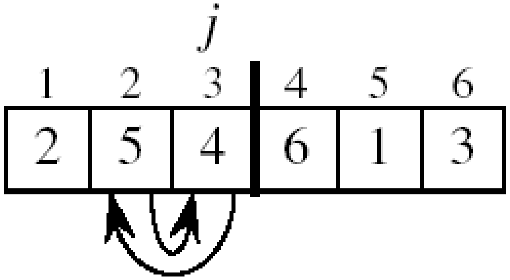
- **Initialization:**
  - Just before the first iteration, $j = 2$:

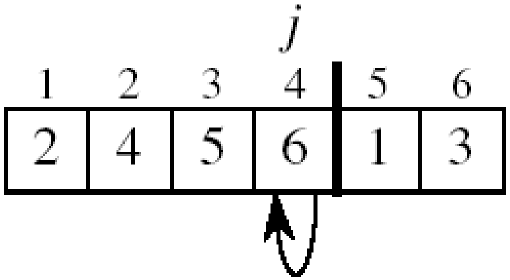
    the subarray $A[1 \,..\, j-1] = A[1]$, (the element originally in $A[1]$) – is sorted



Loop Invariant for Insertion Sort

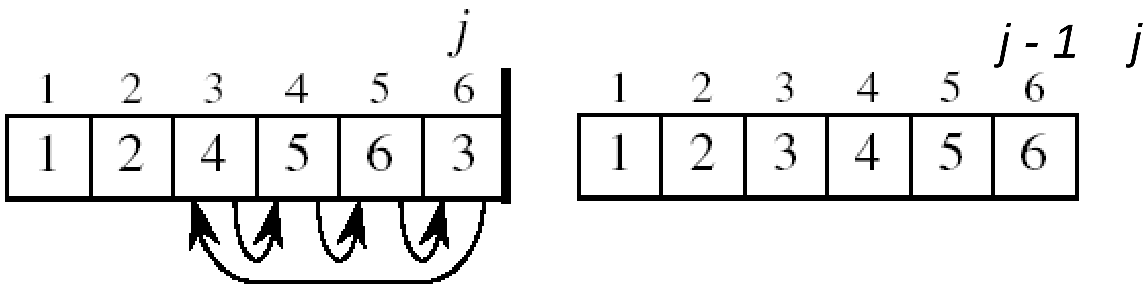
- **Maintenance:**
  - the **while** inner loop moves A[j -1], A[j -2], A[j -3], and so on, by one position to the right until the proper position for key (which has the value that started out in A[j]) is found
  - At that point, the value of key is placed into this position.





Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order



- The enti

Invariant: at the start of the **for** loop the elements in $A[1 \,..\, j-1]$ are in sorted order

Analysis of Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	$n-1$
▷ Insert A[j] into the sorted sequence A[1 . . j -1]	0	$n-1$
i ← j - 1	c_4	$n-1$
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	$n-1$

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} \left(t_j - 1\right) + c_7 \sum_{j=2}^{n} \left(t_j - 1\right) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted

"while i > 0 and A[i] > key"

  - A[i] ≤ key upon the first time the **while** loop test is run (when *i* = *j* -1)
  - $t_j = 1$

- $T(n) = c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}\left(t_j - 1\right) + c_7\sum_{j=2}^{n}\left(t_j - 1\right) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position $\Rightarrow$ compare with j-1 elements $\Rightarrow$ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT(*A*)	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

$\approx n^2/2$ comparisons

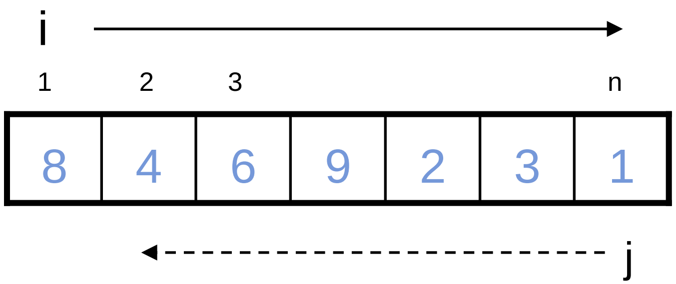
$\approx n^2/2$ exchanges

Insertion Sort - Summary

- Advantages
  - Good running time for “almost sorted” arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges

BUBBLE SORT (EX. 2-2, PAGE 38)

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

EXAMPLE

8	4	6	9	2	3	1

i = 1 j

8	4	6	9	2	1	3

i = 1 j

8	4	6	9	1	2	3

i = 1 j

8	4	6	1	9	2	3

i = 1 j

8	4	1	6	9	2	3

i = 1 j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

i = 6 j

1	2	3	4	6	8	9

i = 7

j

BUBBLE SORT

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

  do for j ← length[A] **downto** i + 1

    do if A[j] < A[j -1]

      i ~~**then** exchange A[j]~~ ↔ A[j-1]

8	4	6	9	2	3	1

i = 1 ←-------------------- j

Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A] c_1

do for j ← length[A] **downto** i + 1 c_2

Comparisons: ≈ $n^2/2$ **do if** A[j] < A[j -1] c_3

Exchanges: ≈ $n^2/2$ **then** exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

$$\text{where } \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, $T(n) = \Theta(n^2)$

Selection Sort (Ex. 2.2-2, page 27)

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

SELECTION SORT

Alg.: SELECTION-SORT(*A*)

8	4	6	9	2	3	1

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*)

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

$\approx n^2/2$ comparisons

$\approx n$ exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$